R Small Group: Class 2

Amy Pomeroy, Dayne Filer, and Sara Taylor

June 20, 2017

Using this document

- Code blocks and R code have a grey background (note, code nested in the text is not highlighted in the pdf version of this document but is a different font).
- # indicates a comment, and anything after a comment will not be evaluated in R
- The comments beginning with ## under the code in the grey code boxes are the output from the code directly above; any comments added by us will start with a single #
- While you can copy and paste code into R, you will learn faster if you type out the commands yourself.
- Read through the document after class. This is meant to be a reference, and ideally, you should be able to understand every line of code. If there is something you do not understand please email us with questions or ask in the following class (you're probably not the only one with the same question!).

Class 2 expectations

1. Understand the basic R data structures (vector, matrix, list, data.frame)
2. Know how to subset the four basic data structures

Introduction to data structures

Vectors The most simple data structure available in R is a vector. You can make vectors of numeric values, logical values, and character strings using the `c()` function. For example:

```
c(1, 2, 3)
## [1] 1 2 3
c(TRUE, TRUE, FALSE)
## [1]  TRUE  TRUE FALSE
c("a", "b", "c")
## [1] "a" "b" "c"
```

You can also join to vectors using the `c()` function.

```
x <- c(1, 2, 5)
y <- c(3, 4, 6)
z <- c(x, y)
z
## [1] 1 2 5 3 4 6
```

Matrices A matrix is a special kind of vector with two dimensions. Like a vector, a matrix can only have one data class. You can create matrices using the `matrix` function as shown below.

```
matrix(data = 1:6, nrow = 2, ncol = 3)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

As you can see this gives us a matrix of all numbers from 1 to 6 with two rows and three columns. The `data` parameter takes a vector of values, `nrow` specifies the number of rows in the matrix, and `ncol` specifies the number of columns. By convention the matrix is filled by column. The default behavior can be changed with the `byrow` parameter as shown below:

```
matrix(data = 1:6, nrow = 2, ncol = 3, byrow = TRUE)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Matrices do not have to be numeric – any vector can be transformed into a matrix. For example:

```
matrix(data = c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE), nrow = 3, ncol = 2)
##      [,1]  [,2]
## [1,] TRUE FALSE
## [2,] TRUE FALSE
## [3,] TRUE FALSE
matrix(data = c("a", "b", "c", "d", "e", "f"), nrow = 3, ncol = 2)
##      [,1] [,2]
## [1,] "a"  "d"
## [2,] "b"  "e"
## [3,] "c"  "f"
```

Like vectors matrices can be stored as variables and then called later. The rows and columns of a matrix can have names. You can look at these using the functions `rownames` and `colnames`. As shown below, the rows and columns don't initially have names, which is denoted by `NULL`. However, you can assign values to them.

```
mat1 <- matrix(data = 1:6, nrow = 2, ncol = 3, byrow = TRUE)
rownames(mat1)
## NULL
colnames(mat1)
## NULL
rownames(mat1) <- c("Row 1", "Row 2")
colnames(mat1) <- c("Col 1", "Col 2", "Col 3")
mat1
##       Col 1 Col 2 Col 3
## Row 1     1     2     3
## Row 2     4     5     6
```

It is important to note that similarly to vectors, matrices can only have one data type. If you try to specify a matrix with multiple data types the data will be coerced to the higher order data class.

The `class`, `is`, and `as` functions can be used to check and coerce data structures in the same way they were used on the vectors in class 1.

```
class(mat1)
## [1] "matrix"
is.matrix(mat1)
## [1] TRUE
as.vector(mat1)
## [1] 1 4 2 5 3 6
```

Lists Lists allow users to store multiple elements (like vectors and matrices) under a single object. You can use the `list` function to create a list:

```
l1 <- list(c(1, 2, 3), c("a", "b", "c"))
l1
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "a" "b" "c"
```

Notice the vectors that make up the above list are different classes. Lists allow users to group elements of different classes. Each element in a list can also have a name. List names are accessed by the `names` function, and are assigned in the same manner row and column names are assigned in a matrix.

```
names(l1)
## NULL
names(l1) <- c("vector1", "vector2")
l1
## $vector1
## [1] 1 2 3
##
## $vector2
## [1] "a" "b" "c"
```

It is often easier and safer to declare the list names when creating the list object.

```
l2 <- list(vec = c(1, 3, 5, 7, 9),
           mat = matrix(data = c(1, 2, 3), nrow = 3))
l2
## $vec
## [1] 1 3 5 7 9
##
## $mat
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
names(l2)
## [1] "vec" "mat"
```

Above the list has two elements, named “vec” and “mat,” a vector and matrix, resepcively.

Data frames Data frames are likely the data structure you will used most in your analyses. A data frame is a special kind of list that stores same-length vectors of different classes. You create data frames using the `data.frame` function. The example below shows this by combining a numeric and a character vector into a data frame. It uses the `:` operator, which will create a vector containing all integers from 1 to 3.

```
df1 <- data.frame(x = 1:3, y = c("a", "b", "c"))
df1
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
class(df1)
## [1] "data.frame"
```

The benefit of data frame objects will be evident in the next section on subsetting objects. Data frame objects do not print with quotation marks, so the class of the columns is not always obvious.

```
df2 <- data.frame(x = c("1", "2", "3"), y = c("a", "b", "c"))
df2
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
```

Without further investigation, the "x" columns in `df1` and `df2` cannot be differentiated. The `str` function can be used to describe objects with more detail than class.

```
str(df1)
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
str(df2)
## 'data.frame':    3 obs. of  2 variables:
##  $ x: Factor w/ 3 levels "1","2","3": 1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Here you see that `df1` is a `data.frame` and has 3 observations of 2 variables, "x" and "y." Then you are told that "x" has the data type integer (not important for this class, but for our purposes it behaves like a numeric) and "y" is a factor with three levels (another data class we are not discussing). ***It is important to note that, by default, data frames coerce characters to factors.*** The default behavior can be changed with the `stringsAsFactors` parameter:

```
df3 <- data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = FALSE)
str(df3)
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: chr  "a" "b" "c"
```

Now the "y" column is a character. As mentioned above, each "column" of a data frame must have the same length. Trying to create a data.frame from vectors with different lengths will result in an error. (Try running `data.frame(x = 1:3, y = 1:4)` to see the resulting error.)

Subsetting objects

We will discuss three subsetting operators: `[`, `[[`, and `$`.

Vectors We will start with vectors, and the `[` operator. First create an example vector, then select the third element.

```
v1 <- c("a", "b", "c", "d")
v1[3]
## [1] "c"
```

The [operator can also take a vector as the argument. For example you can select the first and third elements:

```
v1 <- c("a", "b", "c", "d")
v1[c(1, 3)]
## [1] "a" "c"
```

Lists Similarly, you can use [to subset a list:

```
l1
## $vector1
## [1] 1 2 3
##
## $vector2
## [1] "a" "b" "c"
l1[2]
## $vector2
## [1] "a" "b" "c"
```

Notice that the result of `l1[2]` is still a list. (Try running `class(l1[2])` or `str(l1[2])` to prove to yourself that the result is in fact list.) What if you want to select the vector the list contains? The [operator allows you to subset on elements of a list, as you would for a vector. The [[operator allows you to extract list elements. If you want to extract "vector1" you can either select it by the index (1, because it is the first element in the list) or, in a named list, you can subset by name.

```
l1[[1]]
## [1] 1 2 3
l1[["vector1"]]
## [1] 1 2 3
```

Note, you can also provide a vector of names to the [operator.

Matrices Now we can discuss how to subset on two-dimensional objects. For each dimension the [operator takes one argument. Vectors, being 1 dimension, take one argument. Matrices and data frames take two arguments, given as `[i, j]` where `i` is the the row and `j` is the column. Recall `mat1`:

```
mat1
##       Col 1 Col 2 Col 3
## Row 1     1     2     3
## Row 2     4     5     6
mat1[2, 1]
## [1] 4
```

You can see that an `i` value of `2` and a `j` value of `1` gave the number in the second row and the first column. You do not have to provide both an `i` and a `j` value, providing only one or the other returns the vector for the given row.

```
mat1[ , 3]
## Row 1 Row 2
##     3     6
mat1[1, ]
## Col 1 Col 2 Col 3
##     1     2     3
```

Like with lists, the matrices can also be subset by name when the matrix has row or column names.

```
mat1[ , "Col 1"]
## Row 1 Row 2
##     1     4
```

When subsetting, R attempts to simplify the data structure. As you see in the examples above, subsetting the matrices results in a vector. When you provide a vector to `i` or `j` the result does not *always* simplify to a vector, but may instead maintain the matrix structure.

```
mat1[c("Row 1"), c("Col 1", "Col 3")] ## Can be simplified to a vector
## Col 1 Col 3
##     1     3
mat1[1:2, 2:3] ## Cannot be simplified to a vector
##       Col 2 Col 3
## Row 1     2     3
## Row 2     5     6
```

The other subsetting operator you need to know for this course is `$`. The `$` operator allows you to select list elements by name. For example, consider `l1` again. `l1` has two elements, named "vector1" and "vector2," and you can select one or the other with the `$` operator.

```
l1$vector1
## [1] 1 2 3
```

Notice that you do not put quotes around the list names when providing them to the `$` operator, and the `$` operator can only take a single name.

Data frames Finally, let us discuss subsetting a data frame. Recall from earlier that a data frame is just a special list, so you can subset a data frame in all the same ways you subset a list. Consider `df3`:

```
df3
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
df3[1]
##   x
## 1 1
## 2 2
## 3 3
df3[[2]]
## [1] "a" "b" "c"
df3$x
## [1] 1 2 3
```

Notice how using the `[` operator differs from `[[` and `$`. (You can think of `$` as shorthand for `[[` when you only want to select one element by name.) As we saw in the list subsetting, using `[` preserved the list structure (or in this case, the data frame structure), unlike the `[[` operator which attempts to simplify the data structure.

Unlike a list, you can also subset data frame like you would a two dimensional matrix using both an `i` and a `j` statement.

```
df3[1, 2]
## [1] "a"
df3[2, ]
##   x y
## 2 2 b
df3[ , 1]
## [1] 1 2 3
```

Notice how subsetting by `i` and `j` alone differ. Every column of a data frame is essentially a vector in a list. So when you select one column, the data structure is simplified to a vector. However, selecting 1 row does not simplify, because different columns may have different data classes and it does not make sense to coerce all of the columns to one data type.

Modifying a subset

Recall how you changed the row and column names of the matrix above. Similarly, you can change the values in an object for just a subset using the assignment operator (`<-`).

For example, consider the following matrix.

```
mat2 <- matrix(data = 1:10, nrow = 2)
mat2
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

Now, change the 6 to 100.

```
mat2[2, 3]
## [1] 6
mat2[2, 3] <- 100
mat2
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4  100    8   10
```

The last thing we will mention is that you can add elements to list obejcts using the `[[` and `$` operators. Consider `l2` from above.

```
str(l2)
## List of 2
##  $ vec: num [1:5] 1 3 5 7 9
##  $ mat: num [1:3, 1] 1 2 3
l2$new_element <- "hello"
l2
## $vec
## [1] 1 3 5 7 9
##
## $mat
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
##
## $new_element
## [1] "hello"
```

There is a lot of information about subsetting and modifying objects that we do not have time to cover in this course, so we encourage you to spend some time experimenting on your own.

Class 2 exercises

These exercises are to help you solidify and expand on the information given above. We intentionally added some concepts that were not covered above, and hope that you will take a few minutes to think through what is happening and how R is interpreting the code.

1. Create a data frame with two columns: "col1" with the letters a to f, "col2" with the numbers 1 to 6, and "col3" with the alternating TRUE and FALSE values. Store the data frame as "mydf."
2. Coerce `mydf` from exercise 1 to a matrix using the `as.matrix` function. Predict how the data will change. What class will the data be? Will there be column names? Row names?
3. Subsetting can also be done with logical vectors the same length of the object or dimension you would like to subset. For example, if you have a list `l` with three elements, `l[c(TRUE, FALSE, TRUE)]` would return a list with elements one and three from `l`. Figure out how to use `mydf` to subset only to rows where 'col3' is `TRUE`. (You should attempt to think of a solution that only requires one line of code. Hint: you can pass elements of an obect to itself.)
4. You can stack subsetting operators next to each other. Using `l2` from above, select the 7 from 'vec.' Now try to select the last two rows from 'mat'. (Again, you should attempt to think of a solution that only requires one line of code for each selection.)
5. Recall how you added an element to `l2` using the `$` operator. Again, a data frame is just a special list. Add a column to `mydf` called "col4" with a vector of your choice.